FROM: AFIT/CI

7 July 1989

SUBJECT: Review of Thesis/Dissertation for Public Release

TO: PA

1. Request you review the attached for public release prior to being sent to DTIC.

2. Reply by indorsement to CI NLT _____.

Ernest A. Haygood

ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer
Civilian Institution Programs

1 Atch.
THESIS 89-041
COCKRELL

1st Ind, AFIT/PA

08 FEB 1990

TO: CI

Approved/~~Disapproved~~ for public release.
Log Number: 89-10-87

Harriet D Moultrie

HARRIET D. MOULTRIE, Capt, USAF
Director, Office of Public Affairs

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: NONE
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFIT/CI/CIA- 89-041
6a. NAME OF PERFORMING ORGANIZATION: AFIT STUDENT AT UNIV OF ALABAMA AT TUSCALOOSA / 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: AFIT/CIA
6c. ADDRESS (City, State, and ZIP Code)	7b. ADDRESS (City, State, and ZIP Code): Wright-Patterson AFB OH 45433-6583
8a. NAME OF FUNDING/SPONSORING ORGANIZATION / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification) (UNCLASSIFIED)
A Parallel Architecture for Real-Time Simulation

12. PERSONAL AUTHOR(S)
Clayton Dale Cockrell

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
THESIS/DISSERTATION	FROM _____ TO _____	1989	65

16. SUPPLEMENTARY NOTATION
APPROVED FOR PUBLIC RELEASE IAW AFR 190-1
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer, Civilian Institution Programs

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
ERNEST A. HAYGOOD, 1st Lt, USAF	(513) 255-2259	AFIT/CI

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

AFIT/CI "OVERPRINT"

41

AUTHOR: Clayton Dale Cockrell

TITLE: A Parallel Architecture for Real-Time Simulation

RANK: 1Lt

BRANCH OF SERVICE: Air Force

COMPLETION DATE: 1989

NUMBER OF PAGES: 65

DEGREE: MSEE

INSTITUTION: University of Alabama at Tuscaloosa, Alabama

ABSTRACT

This thesis is concerned with the development of a very fast and highly efficient parallel computer architecture for real-time simulation of continuous systems. Currently, several parallel processing systems exist that may be capable of executing a complex simulation in real-time. These systems are examined and the pros and cons of each system discussed. The thesis then introduces a custom-designed parallel architecture based upon The University of Alabama's OPERA architecture. Each component of this system is discussed and rationale presented for its selection.

The problem selected, real-time simulation of the Space Shuttle Main Engine for the test and evaluation of the proposed architecture, is explored, identifying the areas where parallelism can be exploited and parallel processing applied. Results from the test and evaluation phase are presented and compared with the results of the same problem that has been processed on a uniprocessor system.

AUTHOR: Clayton Dale Cockrell

TITLE: A Parallel Architecture for Real-Time Simulation

RANK: 1Lt

BRANCH OF SERVICE: Air Force

COMPLETION DATE: 1989

NUMBER OF PAGES: 65

DEGREE: MSEE

INSTITUTION: University of Alabama at Tuscaloosa, Alabama

ABSTRACT

This thesis is concerned with the development of a very fast and highly efficient parallel computer architecture for real-time simulation of continuous systems. Currently, several parallel processing systems exist that may be capable of executing a complex simulation in real-time. These systems are examined and the pros and cons of each system discussed. The thesis then introduces a custom-designed parallel architecture based upon The University of Alabama's OPERA architecture. Each component of this system is discussed and rationale presented for its selection.

The problem selected, real-time simulation of the Space Shuttle Main Engine for the test and evaluation of the proposed architecture, is explored, identifying the areas where parallelism can be exploited and parallel processing applied. Results from the test and evaluation phase are presented and compared with the results of the same problem that has been processed on a uniprocessor system.

THE UNIVERSITY OF ALABAMA
THE GRADUATE SCHOOL
OFFICE OF THE DEAN

March 31, 1989

Mr. Clayton Dale Cockrell
c/o Dr. Jon G. Bredeson
The University of Alabama
Tuscaloosa, Alabama 35487

Dear Mr. Cockrell:

This is to inform you that your thesis has been approved. May I congratulate you on the completion of this requirement for your degree program and wish you success in your future professional endeavors.

Sincerely,

William H. Macmillan
Dean

WHM/kn

cc: Dr. Jon G. Bredeson

GRADUATE SCHOOL
UNIVERSITY OF ALABAMA
UNIVERSITY, ALABAMA

TO THE DEAN OF THE GRADUATE SCHOOL:

We, the undersigned, report that as a committee we have examined

NAME: Clayton D. Cockrell STUDENT NO.: 423-78-6913

upon the work done in the subjects assigned, namely:

Major Electrical Engineering

Thesis "A Parallel Architecture for Real-Time Simulation"

and find that h is attainments (are such (~~are not such~~ that he may be

recommended for the degree of Master of Science in Electrical Engineering

Chester C. Carroll

Joseph A. Negus

James E. Lindstrom

I dissent from the foregoing report.

Date February 27, 1989

A PARALLEL ARCHITECTURE FOR REAL-TIME SIMULATION

by

CLAYTON DALE COCKRELL

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in
the Department of Electrical Engineering
in the Graduate School of
The University of Alabama

Tuscaloosa, Alabama

1989

Submitted by Clayton Dale Cockrell in partial fulfillment of the requirements for the degree of Master of Science specializing in Electrical Engineering.

Accepted on behalf of the Faculty of the Graduate School by the thesis committee:

Dr. Joseph Neggers

Dr. James E. Dudgeon

Dr. Chester C. Carroll Chairperson

Dr. Jon G. Bredeson Department Head

Date 2-28-89

Dr. William H. Macmillan Dean, Graduate School

Date 3-30-89

LIST OF ABBREVIATIONS

ACSL	Advanced Continuous Simulation Language
CCV	Coolant Control Valve
CIU	Communication Interface Unit
CPU	Central Processing Unit
DOD	Department of Defense
ET	External Tank
FNCOOL	Fixed Nozzle Cooling
FPU	Floating Point Unit
FPV	Fuel Preburner Valve
HPFP	High Pressure Fuel Pump
HPFT	High Pressure Fuel Turbine
HPOP	High Pressure Oxidizer Pump
HPOT	High Pressure Oxidizer Turbine
IU	Integer Unit
LH2	Liquid Hydrogen
LO2	Liquid Oxygen
LPFT	Low Pressure Fuel Turbine
LPOT	Low Pressure Oxidizer Turbine
MCCOOL	Main Chamber Cooling
MIMD	Multiple-Instruction Multiple-Data
MOV	Main Oxidizer Valve
MPV	Main Fuel Valve
NASA	National Aeronautics and Space Administration

OPERA	Optimally Parallel Environment for Real-Time Applications
OPV	Oxidizer Preburner Valve
PE	Processing Element
RISC	Reduced Instruction Set Computer
SIMD	Single-Instruction Multiple-Data
SSME	Space Shuttle Main Engine
VLSI	Very Large Scale Integrated Circuits

ACKNOWLEDGEMENTS

I am dedicating this thesis to my wife, Konnie Cockrell, and my children, Clay, Lisa, and Lori. Their patience, understanding, and encouragement made the successful completion of this thesis possible.

I would like to express my deepest appreciation to Dr. Chester C. Carroll for providing technical support and guidance throughout my graduate program at The University of Alabama and for agreeing to be my graduate advisor and chairperson of my thesis committee.

I would also like to thank Dr. James E. Dudgeon and Dr. Joseph Neggers for agreeing to be members of my thesis committee, for taking the time and effort necessary to review this manuscript, and for providing helpful comments concerning its content.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis is concerned with the development of a very fast and highly efficient parallel computer architecture for real-time simulation of continuous systems. Currently, several parallel processing systems exist that may be capable of executing a complex simulation in real-time. These systems are examined and the pros and cons of each system discussed. The thesis then introduces a custom-designed parallel architecture based upon The University of Alabama's OPERA architecture. Each component of this system is discussed and rationale presented for its selection.

The problem selected, real-time simulation of the Space Shuttle Main Engine for the test and evaluation of the proposed architecture, is explored, identifying the areas where parallelism can be exploited and parallel processing applied. Results from the test and evaluation phase are presented and compared with the results of the same problem that has been processed on a uniprocessor system.

CHAPTER 1

INTRODUCTION

Advancements in simulations have long been driven by the Department of Defense (DOD), the United States Space Program (directed by the National Aeronautics and Space Administration [NASA]), and their related contractors. The use of actual equipment to test new aircraft, missiles, or space boosters is not only prohibitive from a cost standpoint, but also from a safety point of view. Large complex simulations have been accomplished using analog or hybrid computers in the past; however, due to recent advancements in very large scale integrated circuits (VLSI), high-speed microprocessors, parallel processing techniques, and lower prices, digital computers are being used for many simulations. In the past, attempts to use digital computers concentrated on increasing the raw speed of the processing element and its associated hardware, a method which not only increases the expense of such a system but its complexity as well. Any additional increase in speed is very small compared to the increase in system price and complexity. The architecture proposed in this thesis includes increasing the speed of the processing element and use of more processing elements in an efficient parallel processing scheme.

The remainder of this thesis is organized in the following manner. Chapter 2 describes the proposed architecture for real-time simulations. The overall system is divided into three subsystems: host/allocator, processing elements, and input/output. Each subsystem is described and several alternatives to the present methods are introduced. Chapter 3 is concerned with the testing of the proposed architecture. The problem chosen, simulation of the Space Shuttle's Main Engine, is very large and complex. If the proposed architecture can process this simulation in real-time, it may be scaled up or down in size to process any simulation in real-time. Chapter 4 presents a rationale for each component selection of the proposed architecture. Some of the items discussed in this chapter are: shared memory versus local memory; single instruction multiple data (SIMD) machines versus multiple instruction multiple data (MIMD) machines; and the architecture of competing communication schemes.

CHAPTER 2

A PARALLEL ARCHITECTURE FOR REAL-TIME SIMULATION

Today's digital computers can accurately simulate most physical systems, but such simulations are very complex and require an excessive amount of time to execute. The simulation thus executes much more slowly than real-time. The current state of technology has advanced to the point where implementing a highly parallel computer system capable of processing most simulations in real-time is now both technologically and economically feasible. This chapter provides the design of such a parallel architecture.

Designing an architecture in which several processors work together in parallel is a logical method of improving the execution speed of continuous simulation. Each processor in this architecture would operate on separate portions of the problem to obtain the overall solution. Such an architecture has the capability of simulating continuous systems much faster than a totally sequential architecture, since the performance of parallel architectures is less affected by the system complexity when compared to a sequential architecture. As the system becomes more complex, the execution times of the sequential structures increase.

2.1 The Overall Architecture

The overall architecture is shown in Figures 2.1 and 2.2 and is composed of three units: the host/allocator; execution unit; and the output unit.

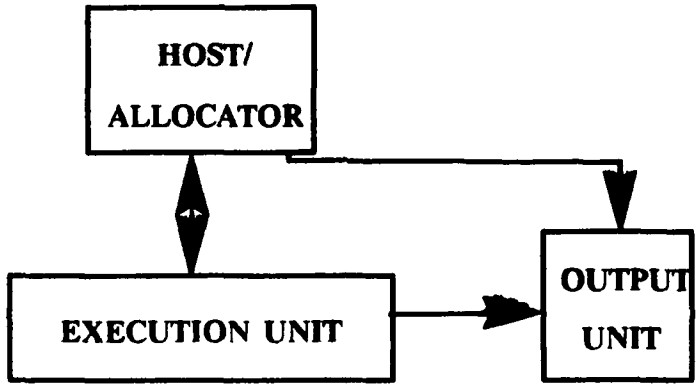


FIGURE 2.1 BLOCK DIAGRAM OF THE OVERALL ARCHITECTURE

The host/allocator is responsible for all preprocessing activity which includes coordinating all input activity and data, executing the serial portions of the simulation, and allocating tasks to the individual clusters. Within the clusters, the intracluster allocator assigns the task to the various processing elements (PE).

The execution stage consists of N clusters with each cluster containing M processing elements. One PE in each cluster is used to allocate the task to the remaining PEs in that cluster. The size

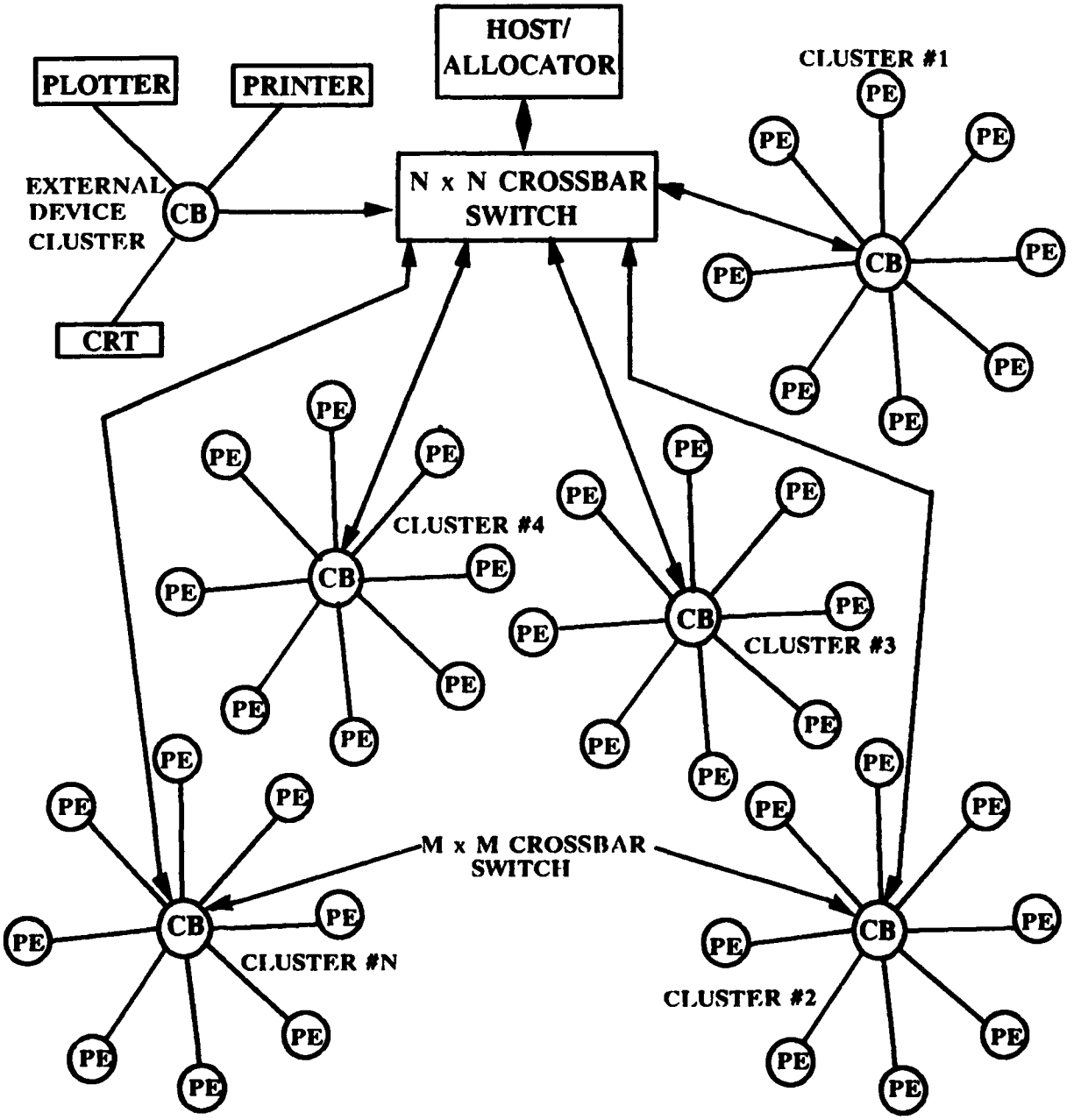


FIGURE 2.2 OVERALL CONFIGURATION

of the crossbar switch is directly proportional to the number of processors and clusters; however, the cost of the crossbar switch determines the size of the overall system, since the price escalates exponentially with the switch complexity.

The output unit is used to transfer the results from the various clusters to any number of external devices.

2.2 The Host/Allocator

The host/allocator may be a conventional computer with a single processing element; however, it would probably be advantageous to have a host with two to four PEs. The serial sections of the simulation are normally at the beginning and end of the simulation process; therefore, if the host contains more than one PE, these sections may also be executed in parallel. The memory size of the host is very important, since the entire continuous simulation program, for each system to be simulated, is stored within it. During the allocation process, portions of this code are allocated by the host to the clusters of the system for parallel execution. The allocation process occurs prior to the actual execution of the program and is dependent upon the integration algorithm and allocation scheme chosen [1].

The allocation process insures that each cluster receives at least one state variable (differential equation) to evaluate. All state variables are assigned their initial values by the host prior to transferring control to the clusters; however, the host may also have to calculate a number of integration points prior to transferring control if the integration algorithm being used is not self-starting. During the execution of the simulation program, the host acts as the input/output point for real-time communications to the outside world.

2.3 The Processing Elements

VLSI technology has advanced such that each node, depending on the amount of memory at each node, may consist of only two to four chips instead of the current seven to twenty chips used in most parallel machines [2, 3]. This not only reduces the size of the overall system but also allows the system to be operated without a special cooling system, thus reducing the cost of the system.

The processing element used as a node in each of the clusters is a state-of-the-art 32-bit processor containing an integer unit (IU) to perform basic processing and a floating point unit (FPU) to perform floating point calculations in parallel with the integer unit. Fully integrating the floating point unit with the rest of

the processor eliminates most of the handshaking and communications overhead that is required if two separate chips are used instead of one. This results in a very large jump in the calculation speed of the processor. The speedup may be as much as three to six times the individual chip speed [4].

The integer unit is the basic processing unit at each node, and it executes all the instructions except the floating point operations. The IU and FPU operate concurrently. When the FPU recognizes a floating point instruction, it is placed in a queue until a floating point register and the required data are available. The IU continues to execute instructions concurrently.

The PE also contains an instruction/data cache memory of 64k-bytes. The cache lowers memory access time from approximately 200-300nS to only 20-50nS [5].

A communication interface unit (CIU) is contained in each PE to control all the information to and from the node. The CIU takes a message from its on-chip processing unit, provides any routing information needed to get that message to the correct destination, and transmits it over the communication network while the processing unit continues to execute instructions. The CIU receives all incoming messages, checks the message (parity check), stores the message in a buffer until it interrupts the processor, and finally transfers the message to the processor. The CIU is connected directly to the processor and local memory via an internal data and address bus. By integrating the CIU directly into the overall PE

chip, it insures very fast communications between the processor and CIU, eliminates additional chips at each node, and insures fast communications between nodes.

2.3.1 Available Technology

A very good alternative to designing a new chip is to use an off-the-shelf design, since it results in not only a faster construction time but also a lower overall system cost. There are several off-the-shelf chips currently available that come close to meeting the above requirements or exceed these requirements. These chips include but are not limited to: the Fairchild Clipper; Intel 80386; Sun's SPARC; and, NCube's custom design chip. Of all the available chips studied, two appear to offer a very good alternative to designing a new chip: the SPARC and the NCube chip.

2.3.2 The SPARC Chip

The scalable processor architecture (SPARC) is a 32-bit reduced instruction set (RISC) type processor that provides both an integer unit and a floating point unit that operate concurrently. The chip

also has an instruction/data cache, a memory management unit and several general and special purpose registers.

The SPARC is supported by about fifty instructions that can be divided into five basic categories:

1. load and store,
2. arithmetic, logic, and shift,
3. coprocessor operations,
4. control transfer, and
5. read and write control.

The SPARC uses an overlapping window scheme ultilizing six windows. Each window is composed of twenty-four working registers and eight global registers [7].

The latest SPARC chip available operates at 50MHz with a 100MHz chip currently being developed.

2.3.3 The NCube Chip

The NCube Corporation of Beaverton, Oregon, manufactures, a line of parallel computers ranging from a four processor card that plugs into an IBM XT to an 1024 node machine. The heart of the NCube system is the custom-designed NCube node processor.

The NCube node processor provides, on a single VLSI chip, a 32-bit central processing unit (CPU) that includes an integer and floating point unit, a memory management unit, and an interprocessor

communications control unit. Figure 2.3 shows the major functional blocks of the chip [6]. The chip contains about 160,000 transistors and has 68 pins.

The node processor has a full set of arithmetic and logical operations on 8-, 16-, and 32-bit integers. It also conforms to the IEEE 754 floating point standard with operations on both 32- and 64-bit real data. There are 16 32-bit long registers, 13 special purpose registers, and a set of addressing modes, including support for vector and matrix operations [8].

2.3.4 Comparison

Both the SPARC and NCube chip have the IU and FPU combined on a single chip, a cache memory, and a memory management unit/communication controller. The big difference is the speed which the two processors operate at and the communication structure of the two chips. A 50MHz SPARC processor is currently available, with a 100MHz version promised in the near future. The current NCube processor operates at 10MHz, but a version eight times as fast is reported to be in the works.

Both the SPARC and NCube chip meet all the minimum requirements set forth at the beginning of this chapter. The NCube chip, although slower than the SPARC chip, has a greater flexibility for parallel

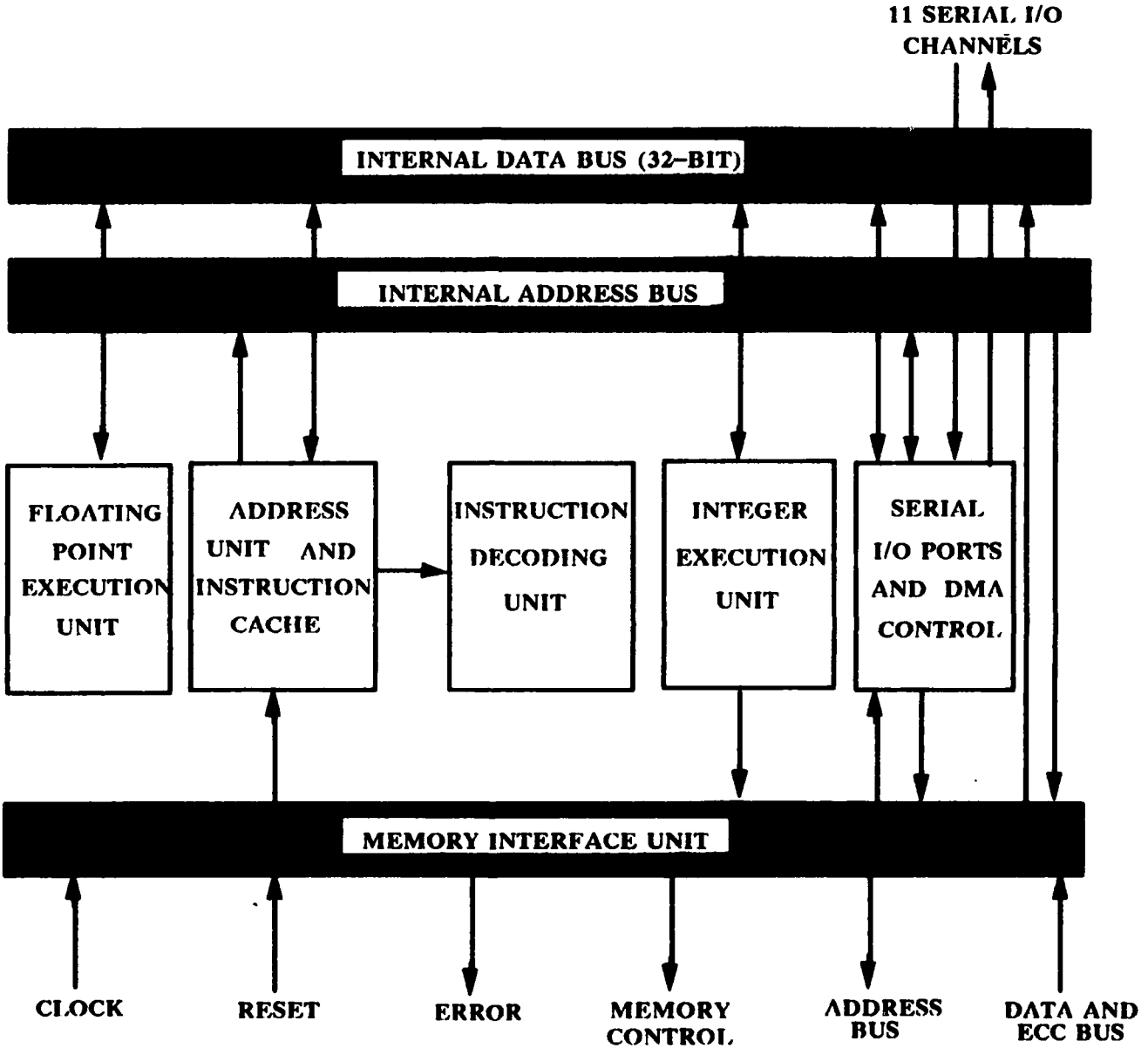


FIGURE 2.3 MAJOR FUNCTIONAL BLOCKS OF A NCUBE PROCESSOR

applications than the SPARC chip due to its greater communications capacity. If the custom design processing element could not be used, the NCube would be the logical choice followed by the SPARC chip.

2.4 Memory

Since there may be hundreds or even thousands of nodes in a parallel computer, their chip count is the most significant part of the total system chip count. Since the processing portion of each node has been reduced to one chip, the memory chip or chips is the next place to try to reduce the overall chip count. By using the densest memory chips currently available, each node can have a 1M-byte local memory ultilizing just two chips. This results in each node in the proposed architecture consisting of only three chips; therefore, a system with 1024 nodes would consist of only 3072 microchips. The NCube 1024 node system contains 7168 microchips.

2.5 Communications

2.5.1 Tokens

The proposed architecture uses data packets (tokens) for all

intracluster and intercluster communications. There are two different types of tokens, data and instructions, each named after the type of information it contains. The instruction packets may contain interrupts, system initiation, procedure calls, or instructions for running a routine.

Each token contains a minimum of four distinct fields as shown in Figure 2.4, allowing each PE, cluster, and external device to have an unique address. The first field identifies to which of the N-clusters the package is to be routed, while the second field identifies the PE or external device within that cluster which is the packet's final destination. The third field contains the specific instruction/data or the address of the instruction/data, and the fourth field determines whether the packet is for data or instructions. Additional fields can be added as the need arises and can be used for a variety of purposes such as error detection, health checks of each node or external device, etc. The remainder of the packet contains the information to be transmitted.

FIGURE 2.4 TOKEN FIELDS

2.5.2 Cluster Configuration

Various configurations such as ring, star, and tree topologies can be used to interconnect the PEs that form the clusters and also to interconnect the clusters. Every PE is connected to every other PE in a completely connected system; therefore, in a system with n PEs, the complete network would require n-square connections [9]. A completely interconnected system becomes prohibitively expensive and complex as the number of PEs in a system increases (n becomes larger); however, it is extremely fast and eliminates all bus conflicts within the system.

The interconnection scheme chosen for the proposed architecture uses a star topology with a crossbar switch as the passive center of the star. The star topology is used for both intracluster and intercluster communications. The maximum distance a message has to travel within a cluster is only two communication paths and for intercluster communications the maximum distance increases to only four communication paths as shown in Figure 2.5 [9].

2.6 Output Unit

The output unit can be any of a variety of currently available output devices. These devices maybe CRTs, printers, plotters, etc.

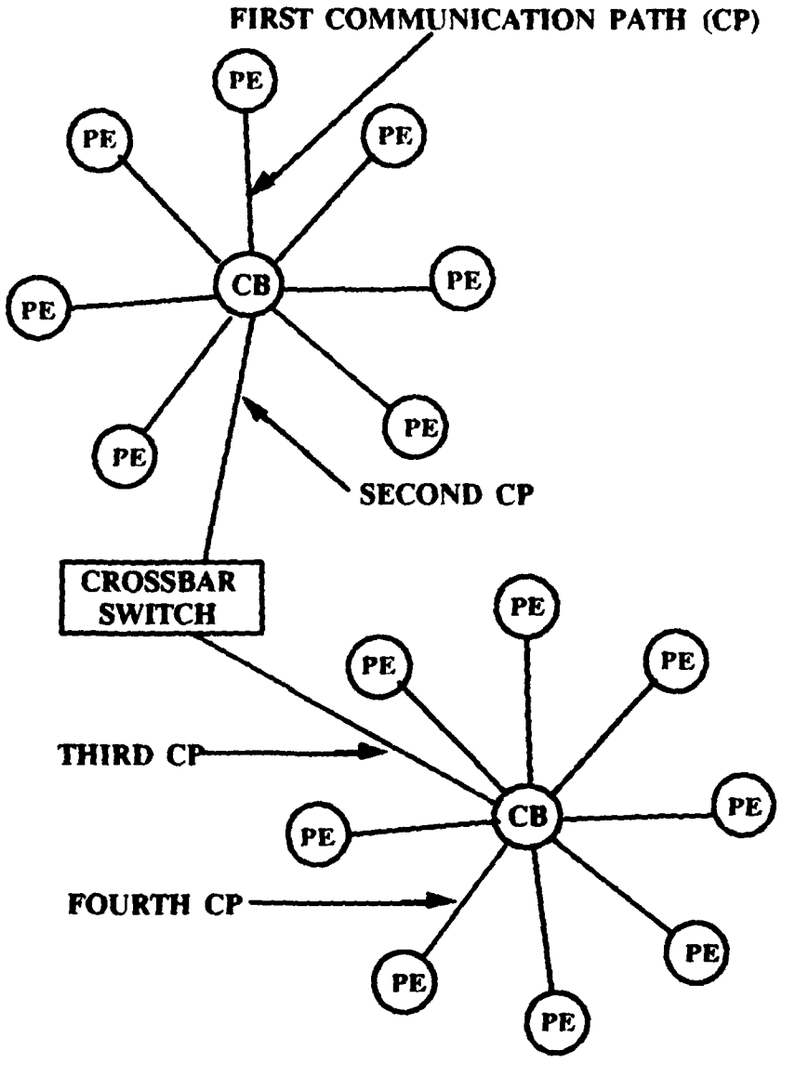


FIGURE 2.5 INTERCLUSTER COMMUNICATION PATH

The proposed architecture treats such devices as just another cluster in the system, allowing for faster and more reliable transfer of data from the various PEs in the system to these external devices. They are connected to the system by a separate crossbar switch as shown in Figure 2.2.

CHAPTER 3

ARCHITECTURE TEST AND EVALUATION: SIMULATION OF THE SPACE SHUTTLE MAIN ENGINE (SSME)

The objective of this chapter is to carefully examine the proposed architecture using the Space Shuttle Main Engine (SSME) simulation and to evaluate its performance as compared to existing uniprocessor systems.

One of the main objectives after designing a new computer architecture is to test it by first emulating the system on an existing machine and then by actually constructing the system. To emulate the proposed architecture, an existing parallel computer with at least 32 nodes would be needed. As of the writing of this thesis, The University of Alabama does not have such a machine available; therefore, a complete test and evaluation of the proposed architecture is not possible at this time.

3.1 SSME Simulation Background

The SSME is a very powerful but sensitive system which requires a carefully controlled rapid transfer of mass and energy during the

first eight minutes of a launch. Real-time simulation of the main engines is required for a variety of design, test, and evaluation purposes. These include testing new or redesigned engine controllers, turbines, and pumps and also evaluating new parameters and constants in the software designed to control the SSMEs. Currently the only way of testing the engines in real-time is by actually firing up the engines on the shuttle or on a test stand.

Earlier successful real-time simulations of the SSME involved hybrid computers with two or more large-scale analog computers, which are now obsolete. The full SSME model has not been solved on a purely digital computer as of the writing of this thesis; however, the SIMSTAR multiprocessor designed by Electronics Associates, Incorporated has came closer than any other system. SIMSTAR utilizes digital, logic, and analog processing to simulate the steady state performance of the SSMEs, using a programming environment based upon the Advanced Continuous Simulation Language (ACSL). Although it has been successful at simulating the steady state performance of the SSME, SIMSTAR has not been able to simulate the transient model of the SSME in real-time.

The SSMEs and their related propellant supply system contain pressures up to 10,000 psia and flows exceeding 1,000 pounds/second resulting in the natural frequencies of the system exceeding 1 kilohertz [10]. An accurate real-time simulation of the SSME over its full operating range is hard to accomplish because of these high natural frequencies and it is a tightly coupled, nonlinear system.

Parallel processing can achieve the computational rates necessary to process this large and complex set of equations, resulting in a real-time simulation.

3.2 The Space Shuttle Main Engine

The Space Shuttle contains three reusable, high-performance, liquid-propellant rocket engines with variable thrust that operate for a total of 480 seconds from launch. The fuel used by the three main engines is liquid hydrogen (LH2), and the oxidizer is liquid oxygen (LO2). The propellant is carried in separate tanks in the external tank (ET) and supplied to the main engines under pressure. The engines can be throttled over a range of 65 to 109 percent of their rated power level in one-percent increments. This provides for high thrust at liftoff and a maximum of 3g acceleration during final ascent [11].

The engine description is divided into primary propellant flow paths between the major subsystems, since the physical structure of the engine is not critical to the simulation [10]. The engine is subdivided into ten major subsystems through which these flows occur. The subsystems are:

1. Low-Pressure Fuel Pump/Turbine
2. High-Pressure Fuel Pump/Turbine
3. Low-Pressure Oxidizer Pump/Turbine

4. High-Pressure Oxidizer Pump/Turbine
5. Combustion Chamber
6. Fuel Preburner
7. Oxidizer Preburner/Boost Pump
8. Coolant Liner Pressure
9. Fixed Nozzle Cooling
10. Main Chamber Cooling.

Five control valves are provided for throttling each engine:

1. Main Fuel Valve (MFV)
2. Main Oxidizer Valve (MOV)
3. Fuel Preburner Valve (FPV)
4. Oxidizer Preburner Valve (OPV)
5. Coolant Control Valve (CCV).

These valves are controlled by a special control computer designed and manufactured by Honeywell.

3.2.1 Primary Fuel/Oxidizer Flow

A simplified flow diagram is shown in Figure 3.1. The ET contains two individual internal tanks for propellant storage. The lower tank contains a maximum of 385,265 gallons of LH2, and the upper tank holds a maximum of 143,351 gallons of LO2.

Oxidizer from the external tank enters the orbiter's LO2 feedline manifold, where it branches into three parallel paths, one to each engine. The low-pressure oxidizer turbopump (LPOT), the first component encountered by the oxidizer, raises the oxidizer pressure from 120 psia to 450 psia. The high-pressure oxidizer turbopump

HPOT) then raises the pressure to 5200 psia, after which the flow splits into several different paths. The main oxidizer path from the HPOT is to the fuel and oxidizer injectors. With the main engine operating at 109%, the total oxidizer flow to the preburners is 105 pounds/second, while the primary flow to the oxidizer injector is 855 pounds/second.

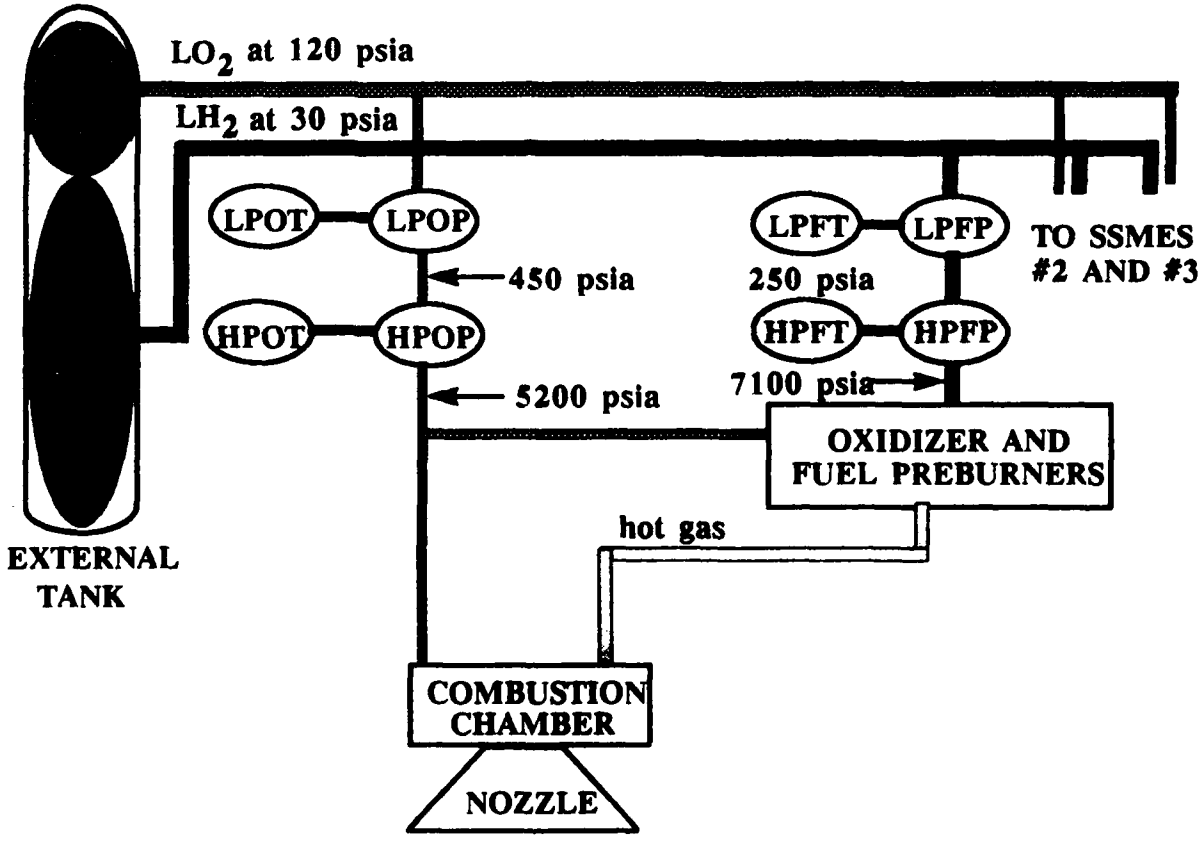


FIGURE 3.1: SSME PRIMARY FUEL/OXIDIZER FLOW

LH2 from the ET enters the orbiter LH2 feedline manifold, where it branches into three parallel lines also. The first component the fuel encounters is the low-pressure fuel turbopump (LPFT), which raises the fuel pressure from 30 psia to 250 psia. After passing through the high-pressure fuel turbopump (HPFT), the fuel pressure reaches 7,100 psia to feed the preburners and fuel injector.

Over the entire engine operating range (65-109%), the oxidizer to fuel ratio is 6:1. At the 109% power level, this produces a flow from the combustion chamber of 1,118 pounds/second at a velocity of 7,500 feet/second, which results in a thrust of 375,000 pounds at lift-off.

3.2.2 Feedback Drive/Control System

To maintain this flow, the SSME employs a feedback drive system utilizing the hot gases produced by the fuel and oxidizer preburners. This process is shown in Figure 3.2. After the fuel passes through the high-pressure fuel pump (HPFP), it goes through the main fuel valve and is divided by the coolant control valve between the fixed nozzle cooling system and the main chamber cooling system. The flow of the fuel through these components serves two purposes: it not only cools the fixed nozzle and the main combustion chamber, but it also raises the temperature of the fuel, giving it a greater

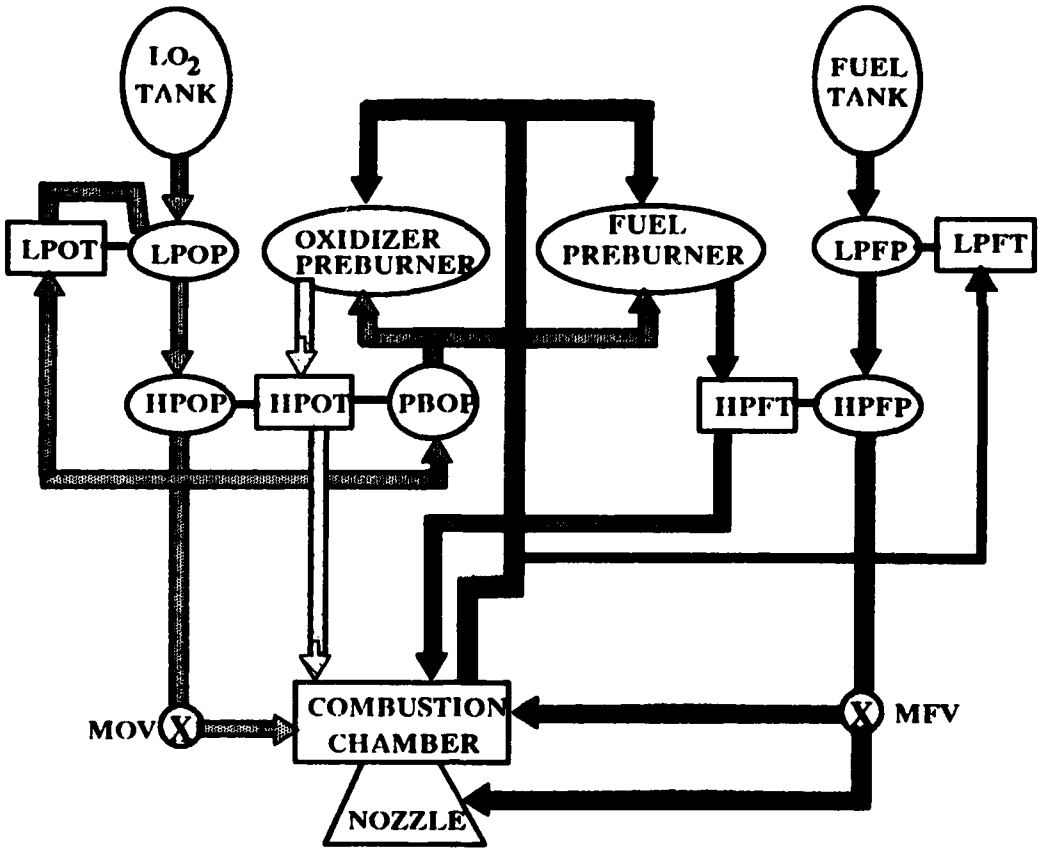


FIGURE 3.2 FEEDBACK DRIVE SYSTEM

potential energy. The heated fuel now makes its way to the preburners. The fuel preburner valve (FPV) and the oxidizer preburner valve (OPV) control the amount of fuel going into the preburner, thus regulating the combustion and pressure within the preburners. The hot gas from the preburners is now used to drive the HPOT and the HPFT. The hot gas pressures in this stream may exceed

5,900 psia. After passing through the high-pressure turbines, the hot hydrogen-rich gas returns to the fuel injector of the main combustion chamber.

The oxidizer flow from the high-pressure oxidizer pump (HPOP) splits into three separate paths. One path goes back to drive the LPOT, while the main flow goes through the MOV to the oxidizer injector in the combustion chamber. The other stream from the HPOP goes through the preburner oxidizer pump and splits into a path to each of the preburners.

The low-pressure fuel turbine and the low-pressure oxidizer turbine are driven by the output of the HPFT and the HPOT, respectively. This is also illustrated in Figure 3.2.

3.3 SSME Mathematical Model

The mathematical model, as supplied by NASA, is composed of 170 equations [12]. They consist of 38 differential equations, 3 difference equations, and 129 algebraic/logical equations. There is a total of 8 input variables, 5 output variables, and 41 state variables. Appendix A contains a listing of the equations.

These equations mathematically model the propellant flows through all ten of the major subsystems of the SSME. For most of the flow paths, the following general form of the equations is used:

Pressure

$$P = k \int [\Sigma DW_i - \Sigma DW_o] dt$$

Flow

$$DW = k_{DW} \int [P_i - P_o - C (DW^2)] dt$$

One subsystem generates a pressure which in turn produces a flow within the next subsystem. The flow information then loops back to the first subsystem, thus closing the loop or circuit. For the heat exchangers, FNCOOL and MCCOOL, the temperatures and densities are also needed to determine heat flux and gas quality. Enthalpy H3 of the fuel entering the heat exchangers from the HPFP is needed to determine the heat transfer [10]. Figure 3.3 is a block diagram of the SSME model showing the flow of information into each subsystem.

3.3.1 Decomposition of Equations into Task Blocks

The next step after obtaining the mathematical model is the decomposition of the equations into task blocks. This allows for efficient load distribution, determination of task execution sequence, and it also helps determine the total number of processors needed. To aid in the decomposition process, certain assumptions

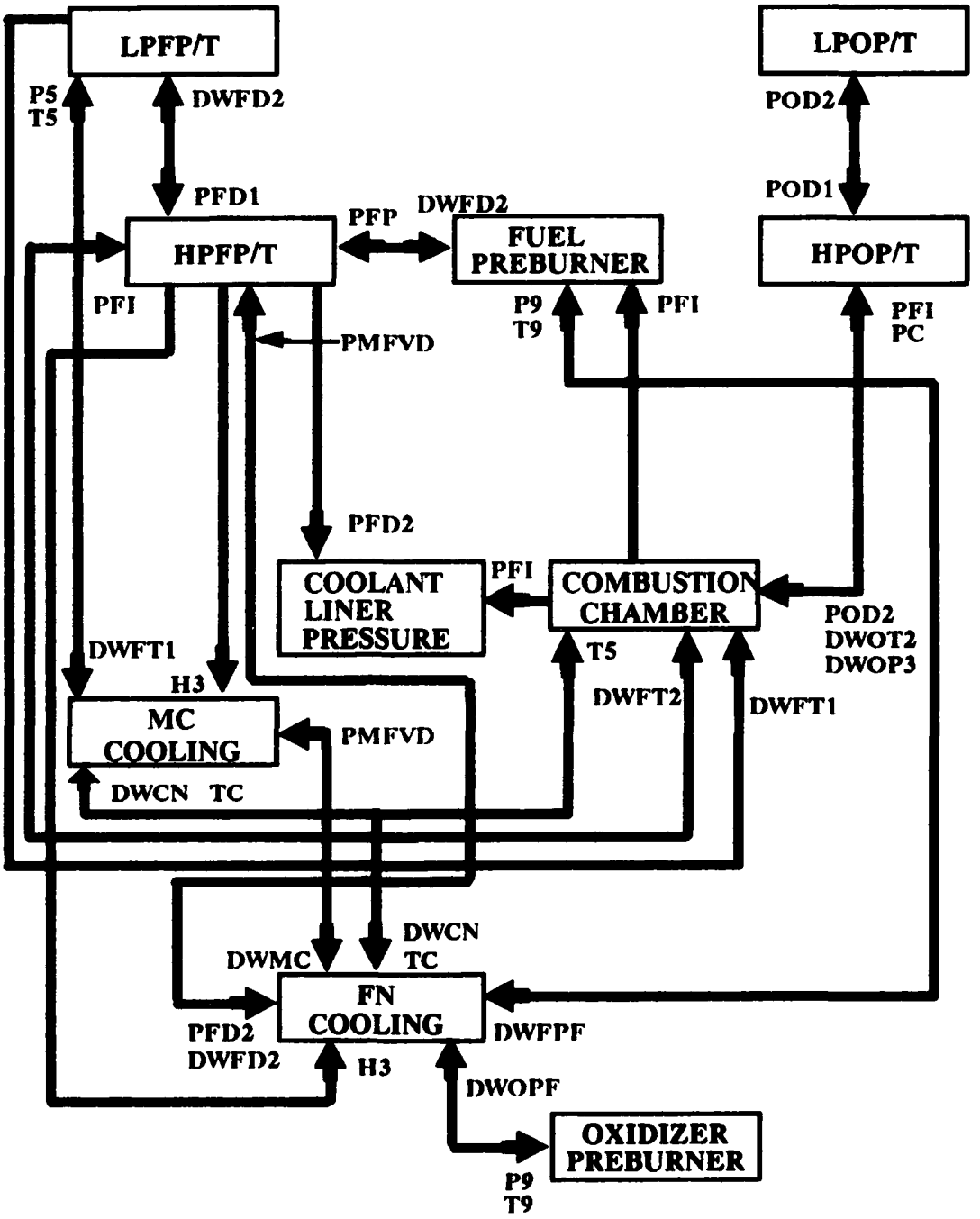


FIGURE 3.3 SUBSYSTEMS FLOW DIAGRAM

have been made:

1. Communication time between processors is considered negligible as compared with the computation time for a task.
2. Operating system overhead is negligible as compared with the computation time for a task.

The model is made up of 170 equations representing the flows between the 10 major subsystems of the SSME. These equations can be decomposed into one of four different models, depending on the level of decomposition:

1. Coarse-grained
2. Medium/Coarse-grained
3. Medium-grained
4. Fine-grained.

The coarse-grained model breaks the equations down into the ten different subsystems and treats each subsystem as a separate task. This results in a total of 10 task blocks in the coarse-grained model.

The medium/coarse-grained model treats each state variable as a separate task. The equations are therefore broken into a total of 41 task blocks which result in some equations appearing in several different task blocks. The duplication of the equations in the different task blocks is unavoidable, since the output of some equations is used as the input to several other equations.

The medium-grained model treats each equation as a task resulting in a total of 170 task blocks. The fine-grained model breaks each equation down into several individual tasks (elementary

mathematical operations) with each task taking approximately one clock cycle to execute.

It is very obvious that the inherent parallelism is the highest in the fine-grained model and the lowest in the coarse-grained model. Whenever a problem can be broken down into separate elementary mathematical operations (add, subtract, multiply, divide, etc.), the potential parallelism in the system increases and the overall processing time is minimized if an efficient scheduling of the task on an appropriate number of processors is achieved. This apparent increase in parallelism makes the fine-grained model very attractive if speed is the only consideration; however, the fine-grained model has some problems because it is broken into these individual mathematical operations. It is harder to change the equations used to model a particular subsystem since these equations have been broken into separate pieces and distributed to separate processing elements. If the simulation is to be interactive, it would be very difficult to use the fine-grained model since any changes in the simulation while it is executing would affect numerous processors within the system. The medium-grained model does not have these problems associated with it; it appears that the medium-grained model can be executed in real-time by using parallel processing.

3.3.2 Task Graph

The medium-grained model contains 170 separate task blocks;

most of these blocks are dependent upon a previous block for at least one of its inputs. In order to determine these dependencies, a software package developed by Earl Wells was used. To use the dependency software, the equations have to be broken down, showing which outputs are used as inputs, which inputs are external to the equations, and what are the state variables. Listed below is an example of how the equations were broken down:

Equation Number	Equation Output	Equation Inputs
1	OFP1	DWFD2 SF1
2	PFD1	PFS SF1 OFP1
3	TFP1	SF1 OFP1

where DWFD2 is the output from equation 17, SF1 is the output from equation 6, and PFS is an external input.

The software package goes through and identifies all the equations that are dependent only on external inputs and/or state variables. These equations are listed as level-one in the task dependency chart. The next trip through it identifies all the equations that are dependent upon the outputs of level-one equations and external inputs or state variables. These equations form level-two of the task dependency chart. The third trip identifies all the equations dependent upon external inputs or state variables and outputs from levels-one and -two. These equations make up level-three of the chart. This process continues until all the equations have been eliminated and assigned to a level in the task dependency chart.

Table 3.1 shows the output of the dependency software. It divided the 170 equations into 10 different levels of dependency.

TABLE 3.1
TASK DEPENDENCY CHART
SHUTTLE MAIN ENGINE MEDIUM-GRAINED MODEL

Level 1 - Number of Task Blocks: 28
EQ17 EQ18 EQ23 EQ36 EQ38 EQ40 EQ42 EQ44 EQ50 EQ55 EQ62 EQ67 EQ83 EQ88 EQ94 EQ99 EQ112 EQ113 EQ117 EQ127 EQ134 EQ146 EQ148 EQ150 EQ160 EQ163 EQ165 EQ170

Level 2 - Number of Task Blocks: 33
EQ1 EQ8 EQ13 EQ14 EQ37 EQ39 EQ41 EQ43 EQ45 EQ46 EQ51 EQ52 EQ57 EQ64 EQ89 EQ100 EQ116 EQ123 EQ128 EQ130 EQ135 EQ136 EQ144 EQ147 EQ149 EQ151 EQ152 EQ158 EQ161 EQ164 EQ166 EQ168 EQ169

Level 3 - Number of Task Blocks: 24
EQ2 EQ3 EQ7 EQ16 EQ22 EQ47 EQ56 EQ68 EQ82 EQ84 EQ86 EQ93 EQ95 EQ97 EQ110 EQ120 EQ129 EQ131 EQ140 EQ142 EQ145 EQ156 EQ159 EQ167

Level 4 - Number of Task Blocks: 25
EQ9 EQ15 EQ24 EQ26 EQ27 EQ48 EQ53 EQ58 EQ63 EQ69 EQ71 EQ72 EQ85 EQ87 EQ96 EQ98 EQ111 EQ114 EQ122 EQ124 EQ125 EQ132 EQ141 EQ143 EQ157

Level 5 - Number of Task Blocks: 15
EQ10 EQ28 EQ34 EQ49 EQ54 EQ65 EQ73 EQ79 EQ92 EQ103 EQ115 EQ121 EQ126 EQ133 EQ162

Level 6 - Number of Task Blocks: 12
EQ11 EQ25 EQ29 EQ35 EQ66 EQ70 EQ74 EQ80 EQ109 EQ118 EQ137 EQ153

Level 7 - Number of Task Blocks: 11
EQ12 EQ30 EQ75 EQ81 EQ90 EQ101 EQ104 EQ106 EQ119 EQ138 EQ154

Level 8 - Number of Task Blocks: 11
EQ4 EQ19 EQ31 EQ59 EQ76 EQ91 EQ102 EQ105 EQ107 EQ139 EQ155

Level 9 - Number of Task Blocks: 6
EQ5 EQ20 EQ32 EQ60 EQ77 EQ108

Level 10 - Number of Task Blocks: 5
EQ6 EQ21 EQ33 EQ61 EQ78

The number of tasks per level ranges from 5 to 33. The task dependency chart could be used to determine the number of processors required for the SSME simulation because the level with the largest number of tasks identifies the minimum number of processors that

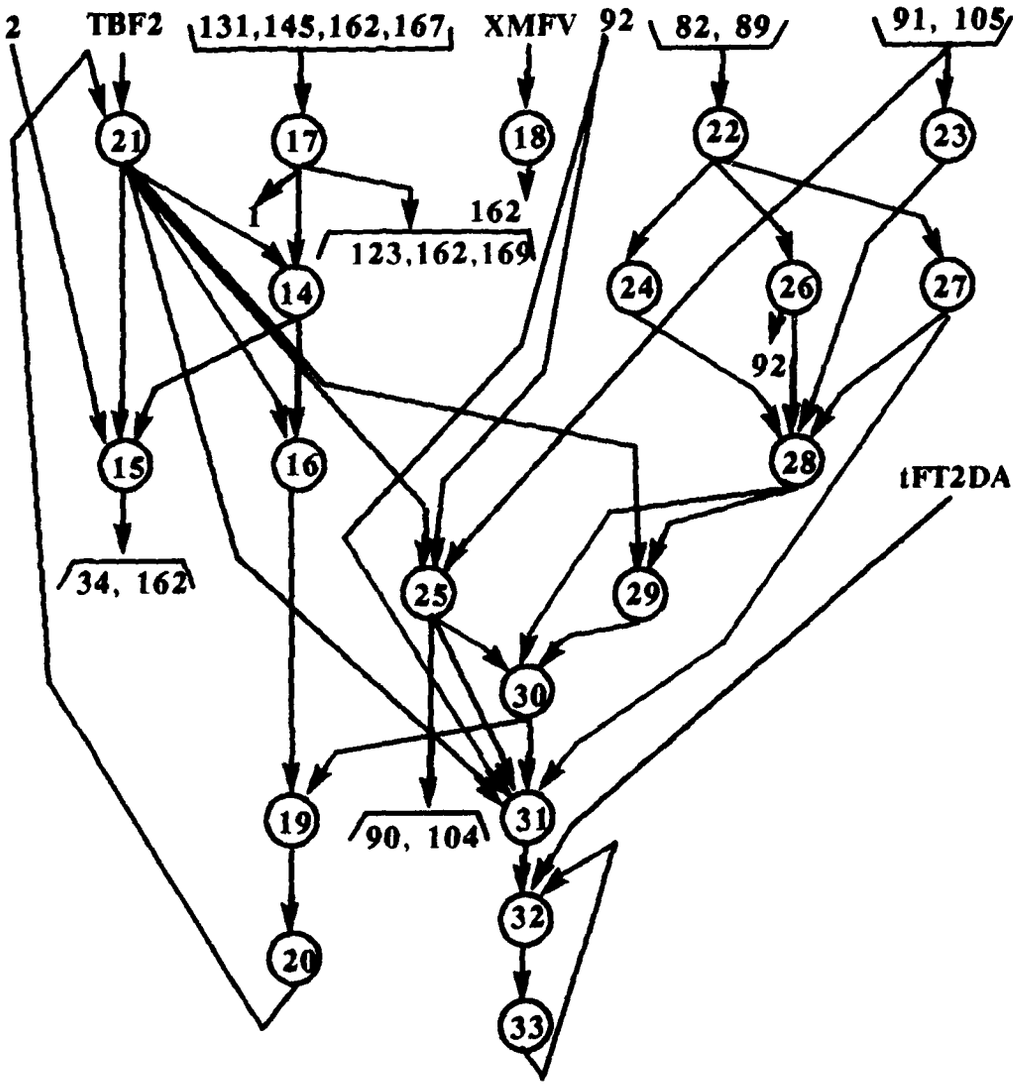
would be required. In this particular case that would imply 33 processors, since level-2 of the chart contains 33 separate task blocks; however, this may not be the optimal number of processors that could be used.

By using the task dependency chart to create a task (directed) graph, a better understanding of the equation flow can be achieved [13]. (The graph will not be included in this thesis due to its size and complexity.) The nodes (vertices) of the graph will represent the equations of the medium-grained model. The arcs between the nodes will represent the data flow (output of an equation) from one equation to the next. The arrow head on the arcs will point to the node receiving the data. Figure 3.4 illustrates a task graph for the HPFP/T subsystem of the SSME.

The task graph pointed out some equations in the task dependency chart that could be moved from one level of the chart down to another. These moves are possible because the outputs of these equations are not used as inputs in the next lower level; instead, they are used as inputs two to seven levels down in the task dependency chart. The rearranged task dependency chart is shown in Table 3.2. The rearrangement results in five fewer processors being needed to implement the task dependency chart. Level-four of the new chart now has the most tasks with 28, where level-3 in the first chart had the most with 33.

The software used does not contain an error. In scheduling tasks on a multiprocessor system, the first priority is to get as

HIGH PRESSURE FUEL PUMP (HPFP) 14–18 HPF TURBINE 22–33
HPF TURBOPUMP SPEED 19–21



1 AND 2 --LPF PUMP
34--COOLANT LINER PRESSURE
82--FUEL PREBURNER FUEL FLOW
89--FUEL PREBURNER OXIDIZER FLOW
90, 91, AND 92--FUEL PREBURNER COMBUSTOR
104--MAIN CHAMBER FUEL
105--INJECTOR
123--THRUST CHAMBER
131--MAIN CHAMBER REGENERATIVE COOLING ELEMENT
145--FIXED NOZZLE REGENERATIVE COOLING ELEMENT
162, 167, AND 169--FIXED NOZZLE INLET, DISCHARGE, AND BYPASS

FIGURE 3.4 TASK GRAPH FOR THE HPFP/T SUBSYSTEM

TABLE 3.2
TASK DEPENDENCY CHART
SHUTTLE MAIN ENGINE MEDIUM-GRAINED MODEL

Level 1 - Number of Task Blocks: 21
EQ17 EQ36 EQ38 EQ40 EQ42 EQ44 EQ50 EQ55 EQ62 EQ88 EQ99 EQ113 EQ127 EQ134 EQ146 EQ148 EQ150 EQ160 EQ163 EQ165 EQ170

Level 2 - Number of Task Blocks: 27
EQ1 EQ13 EQ14 EQ37 EQ39 EQ41 EQ43 EQ45 EQ46 EQ51 EQ83 EQ89 EQ94 EQ100 EQ116 EQ130 EQ135 EQ136 EQ144 EQ147 EQ149 EQ151 EQ152 EQ158 EQ161 EQ166 EQ168

Level 3 - Number of Task Blocks: 26
EQ2 EQ7 EQ8 EQ22 EQ47 EQ52 EQ56 EQ68 EQ82 EQ84 EQ86 EQ93 EQ95 EQ97 EQ110 EQ112 EQ120 EQ128 EQ129 EQ131 EQ140 EQ142 EQ145 EQ156 EQ159 EQ167

Level 4 - Number of Task Blocks: 28
EQ9 EQ15 EQ18 EQ23 EQ24 EQ26 EQ27 EQ48 EQ53 EQ63 EQ67 EQ69 EQ71 EQ72 EQ85 EQ87 EQ96 EQ98 EQ111 EQ114 EQ122 EQ124 EQ125 EQ132 EQ141 EQ143 EQ157 EQ164

Level 5 - Number of Task Blocks: 15
EQ10 EQ28 EQ34 EQ49 EQ54 EQ65 EQ73 EQ79 EQ92 EQ103 EQ115 EQ121 EQ126 EQ133 EQ162

Level 6 - Number of Task Blocks: 12
EQ11 EQ25 EQ29 EQ35 EQ66 EQ70 EQ74 EQ80 EQ109 EQ118 EQ137 EQ153

Level 7 - Number of Task Blocks: 15
EQ3 EQ12 EQ16 EQ30 EQ57 EQ64 EQ75 EQ81 EQ90 EQ101 EQ104 EQ106 EQ119 EQ138 EQ154

Level 8 - Number of Task Blocks: 12
EQ4 EQ19 EQ31 EQ59 EQ76 EQ91 EQ102 EQ105 EQ107 EQ117 EQ139 EQ155

Level 9 - Number of Task Blocks: 6
EQ5 EQ20 EQ32 EQ60 EQ77 EQ108

Level 10 - Number of Task Blocks: 8
EQ6 EQ21 EQ33 EQ58 EQ61 EQ78 EQ123 EQ169

large a number of tasks executing as fast as possible. This exploits any parallelism inherit in the system. The software package

accomplishes this by using the dependencies of the equations to set up the task dependency chart. The rearranging of the first chart does not increase the overall execution time of the system; instead, it results in a higher utilization rate for the final system.

The SSME mathematical model contains several equations that involve integration. These equations can be integrated using any one of several parallel integration techniques that have been developed over the past few years. These parallel integration algorithms are considerably faster than the conventional sequential algorithms that have been used in the past. Some of the most frequently used parallel integration algorithms are: the Runge-Kutta method, the Predictor Corrector method, and the Interpolation method.

The speed and accuracy of the integration algorithm used in the simulation will determine the validity of the overall model. This thesis does not attempt to identify the algorithm that is best for this particular application, since nearly any parallel integration algorithm could be implemented on the proposed architecture.

3.4 Verification of Performance

Since the introduction of multiprocessor systems, parameters have been created to measure the performance of these systems. The parameter that is used the most is called the speed-up ratio. The

speed-up ratio is defined as the ratio of the serial execution time to the parallel execution time for the same problem:

speed-up = serial time/parallel time.

In calculating the serial and parallel execution times for this thesis, the actual processing times of the NCube custom design processor operating at 10MHz are used. The processing times are:

multiply	3.4 microsecond (uS)
divide	6.3 uS
square root	6.3 uS
add	2.9 uS

There were also some processing times that still had to be estimated. These items and their estimated execution time on an NCube are:

integration fourth-order Runge-Kutta	15.1 uS
noninteger power functions	258.1 uS

To calculate the serial execution time, the execution time for each primitive mathematical operation in each equation was added together to give a total execution time for that equation. These times were then added together for all 170 equations; this resulted in a total serial execution time of 5,064.1 uS.

The parallel execution time is calculated by first looking at the task graph and locating the critical path (longest path) in the graph. The task graph of the medium-grained model shows that the critical path involves an equation from each of the ten levels,

resulting in a total execution time of 457.7 uS when all the primitive operations are added together. This gives a theoretical maximum speed-up of:

$$\text{speed-up} = 5064.1/457.7 = 11.06.$$

Another important parameter used in measuring multiprocessor performance is the efficiency rate. It is defined as the ratio of the speed-up to the number of processors used:

$$\text{efficiency (e)} = \text{speed-up/number of processors.}$$

The efficiency rate for the medium-grained model is:

$$e = 11.06/28 = .395.$$

The efficiency rate for a uniprocessor system is 1.0 but drops to .395 when 28 processors are used. This does not mean the multiprocessor system is not a good design. A lower efficiency rate is a necessary trade-off if a real-time simulation is to be realized.

A higher speed-up cannot be achieved if fewer than 28 processors are used, since this would increase the number of levels in the task dependency chart. Neither can a higher speed-up occur if more than 28 processors are used, since the critical path in the task graph involves at least one equation from each of the ten levels.

CHAPTER 4

DESIGN CONSIDERATIONS

Whenever someone introduces a new concept or a new design, the first question asked by most people is, why did you do it this way? This chapter attempts to answer some of these whys.

4.1 Questions

The design and development of a parallel processing system raises several questions that must be answered before proceeding with the design. These basic questions are:

1. How powerful should each processor be?
2. How should the processing elements communicate with each other?
3. How should the workload be divided among the processing elements?
4. How can we be sure no processing element sits idle waiting for data?
5. What type of memory do we need and how large should it be?
6. Should the system be a SIMD or a MIMD system?

The remainder of this chapter is devoted to answering each of

these questions. The first section discusses why a parallel architecture was chosen over a sequential architecture.

4.2 Parallel versus Sequential

Simulation is defined in [14] "as a numerical technique for conducting experiments on a digital computer that involves certain types of mathematical and logical models that describe the behavior of a system over extended periods of real time." Simulation currently is the only method for estimating the performance of a new system prior to the actual manufacturing of that system. The demand for real-time simulations is growing at a tremendous rate; this is mainly because of savings realized by detection of faulty designs prior to manufacturing the product. The size and complexity of these simulations are also increasing dramatically, not only because of the size and complexity of the systems they simulate, but also to the demand that the simulations be as realistic and accurate as possible.

Sequential computers cannot achieve processing speeds high enough to run these complex simulations in real time. Charles L. Seitz said in [15], "Whenever a computer designer has reached for a level of performance beyond that provided by his contemporary technology, parallel processing has been his apprentice." This was not only true in 1984 but continues to be the case today, not

because it is necessarily easier to use parallel processing, but because it is the only way to achieve the processing speeds required for these applications.

4.3 System Architecture

4.3.1 SIMD versus MIMD

One of the first tasks is deciding if the system will be a single-instruction multiple-data (SIMD) or a multiple-instruction multiple-data (MIMD) system. In SIMD systems, a single instruction controls all the processors while they operate on different data. Although this system is simpler than a MIMD system, it is not as flexible. The SIMD system cannot accommodate multiple independent users; however, the MIMD system can since each processor is independent of the others and executes its own program [16].

4.3.2 Shared or Local Memory

The second decision to be made is between a shared memory system and a local memory system. In a shared memory system, each processor

must access a single large memory bank. This system has many problems associated with it that are very difficult to overcome. One problem is that the processor to memory bus is easily saturated when multiple processors try to access memory at the same time. Another problem occurs whenever several processors try to access the same segment of memory at the same time; this results in some type of hardware or software protocols being added for arbitration. These conditions result in a bottleneck that tends to slow the overall processing speed of the entire system. A shared memory system is also very hard to scale up to a large number of processors due to the cost and complexity of the switching hardware involved in such a system.

A local memory system is faster than a shared memory, since the local memory is located adjacent to the processing elements, thus eliminating a long data bus or switching network. The local memory also has no inherent limit on the number of processors in a system, since there is not any complex hardware associated with the addition of a few more processing elements.

4.3.3 Power of the Processing Elements

Whatever processing element is used in the overall multiprocessor system, it should be the fastest state-of-the-art

processor available at the time that meets the needs of the proposed architecture. As discussed earlier in Chapter Two, there are several processing elements currently available that meet the above criteria.

4.3.4 Load Distribution

How to efficiently divide the various tasks among the different processing elements is a question that has not been answered completely yet. Research continues on this subject at several major universities both in Europe and here in the United States. In fact two graduate students (Jeff Jackson and Michael Whiteside) here at the University of Alabama are conducting some very promising research into algorithms that will look at the incidence matrix of a data flow graph and select an efficient allocation of the task to the available processing elements.

This thesis does not attempt to allocate the various tasks of the SSME simulation to the processing elements. The only method currently available is to allocate the task by dividing the task graph into clusters that minimize the intercluster communications. The result is that the task graph is mapped or embedded into the machine graph or structure [18]. At the present time, this process must be accomplished manually.

4.3.5 Idle Time

Does it really matter if some processing elements sit idle for a short time waiting for data? That is the question that really needs to be answered next. The answer depends on the desired objectives of the total design. Is the objective to design a system with 100% or near 100% utilization of every processor, or is the objective to design a very high-speed system obtaining as high an utilization rate as possible but still meet all the speed requirements? The proposed architecture places high speed as the first priority while desiring to achieve as high a utilization rate as possible.

4.3.6 Communications

The most important factor, after processor speed, affecting the performance of a multiprocessor system is the interprocessor communications scheme. A multiprocessor system built with the fastest processors available will still have its overall performance degraded dramatically unless it has a very efficient communications scheme. A high-performance efficient communication system insures that whenever a processor needs to communicate with another processor that it can do so without any unnecessary delays and by interrupting only the destination processor.

There are many different ways of organizing an efficient communications system. In [17] it is suggested that, in general, each processor should have just a very few communications links, a small interprocessor distance, and a large number of alternate paths between a pair of processors for fault tolerance. A small interprocessor distance is a very logical ideal. By keeping the distance between the processors to a minimum, the propagation delay between processors that are not nearest neighbors (next to each other) is also kept to a minimum. If each processor located between the source and destination processors must be interrupted to transmit a message, then the suggestion that each processor have only a very few communication lines makes good sense. If each processor had more than one or two lines, it would not really serve any useful purpose, since the processor would have to halt its computation task each time a message was being passed by it, to it, or from it. However, if the interprocessor communications are transparent to the individual processor, as suggested in Chapter Two for the proposed architecture, then each processor could have six, eight, ten, or more outside communications links. This would virtually assure that there would be an open path between any two processors at any given moment of time.

In [16] it is stated that the middle ground in parallel architecture belongs to switch-based system while the two extremes are represented by buses and cubes. The bus architecture has already been discussed in this chapter and discarded due to the many problems

and expense associated with it. The cube architecture has several advantages to it, along with the switch-based system. Both systems can support multiple communications lines at each node if the node processor can accept them. Both systems can have some degree of fault tolerance built into them. The cube architecture already has some degree of fault tolerance in it, since each node in a cube has more than one path to it. The biggest difference between the two architectures occurs when calculating the interprocessor communications distance. In the cube architecture this distance depends upon the order of the cube (a power of 2 representing the size of the cube). The longest communications path for any order hypercube is (log2)(N) where N is the order of the cube. For a simple order 3 node as shown in Figure 4.1, this distance can be as great as 3 communications paths in going from node (0,0,0) to node (1,1,1). This compares to a maximum of 4 communications paths that must be transversed to get to any node in the proposed architecture which is switch-based. The communications paths for this system are shown in Figure 2.5.

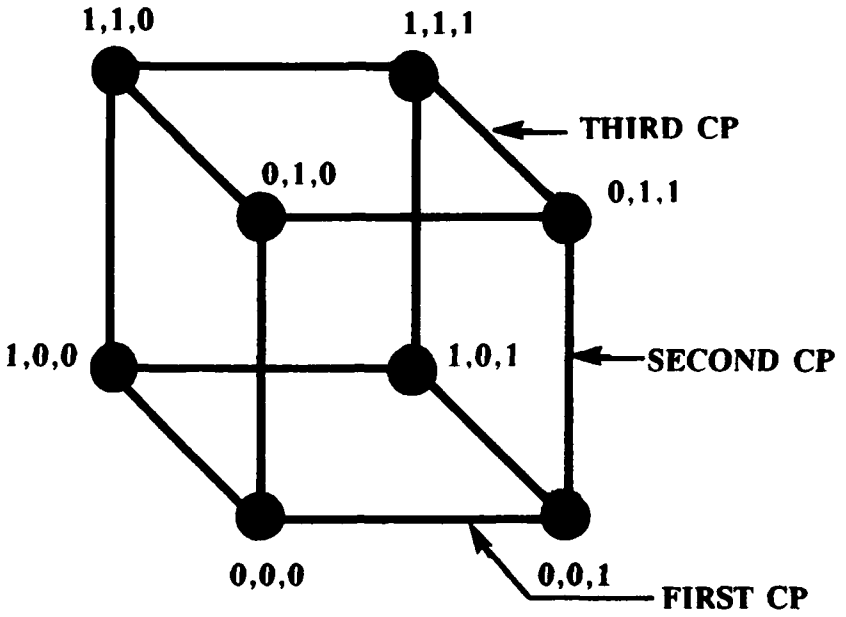


FIGURE 4.1: COMMUNICATION PATHS IN AN ORDER THREE HYPERCUBE

CHAPTER 5

CONCLUSIONS

To meet the growing demand for real-time simulations, an efficient, easy-to-use, inexpensive, flexible parallel architecture must be developed and made widely available. The development of such an architecture is a complex, time-consuming process. The developer must be knowledgeable of not only the hardware and software available, but also the size of the simulation that will be processed on the system and the many problems of processing a continuous simulation in real-time.

To insure a real-time simulation, every available means of using parallelism must be considered. Most large scale simulation problems contain functions that may be executed in parallel. With additional effort many more areas of parallelism can be exploited from these problems. Then by implementing the simulation with parallel algorithms more time can be saved.

Finally, the simulation must be mapped to a parallel architecture that will fully exploit all the parallelism in the simulation and make full use of all parallel algorithms. The proposed architecture discussed within this thesis is such an architecture. It is inexpensive, since it can be designed using off-

the-shelf hardware; it is easy to use, since nearly any computer can serve as the front end for programming and input/output purposes. It is efficient, since it can exploit every little bit of parallelism in most problems, and it is also flexible, since more processors may be added at any time in the future as needed.

By combining this proposed architecture with the available parallel processing techniques, an efficient parallel processing environment has been created that will perform most continuous simulations in real time.

REFERENCES

[1] C. C. Carroll and Buren Earl Wells, "An Intelligent Processing Environment for Real-Time Simulation", Bureau of Engineering Research, Report No. 421-17, The University of Alabama, Tuscaloosa, Al, May 1988.

[2] iPSC Data Sheet, order number: 280110-001, Intel Scientific Computers.

[3] NCube Ten-an Overview, NCube Corp.

[4] David Jurasek, William Richardson, and Doran Wilde, "A Multiprocessor Design in Custom VLSI", VLSI Systems Design, June 1986, pp. 26-30.

[5] Alan Jay Smith, "Cache Memory Design: an Evolving Art", IEEE Spectrum, Vol. 24, No. 12, December 1987, pp. 40-44.

[6] John P. Hayes, Trevor Mudge, and Quentin F. Stout, "A Microprocessor-based Hypercube Supercomputer",IEEE Micro, October 1986, pp. 6-17.

[7] "The SPARC Architecture Manual", Sun Microsystems, Inc., August 1987.

[8] Jacob Barhen and John F. Palmer, "The Hypercube in Robotics and Machine Intelligence", Computers in Mechanical Engineering, March 1986.

[9] Kai Hwang and Faye A. Briggs, Computer Architecture and Parallel Processing, New York, New York: McGraw-Hill Book Company Inc., 1984.

[10] J. Paul Landauer, "Real-Time Simulation of the Space Shuttle Main Engine on the SIMSTAR Multiprocessor", Simulation Computers Application Report, Electronics Associates, Inc., February 1988.

[11] Space Shuttle Transportation System, Rockwell International Corp., January 1984.

[12] Space Shuttle Main Engine Mathematical Model, Revision H, Version 1.3, November 20, 1987.

[13] Claude Berge, Graphs, New York, New York: Elsevier Science Publishing Company, Inc., 1985.

[14] William G. Bulgren, Discrete System Simulation, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1982.

[15] Charles L. Seitz, "Concurrent VLSI Architectures", IEEE Transactions on Computers, Vol. c-33, No. 12, December 1984, pp. 1247-1265.

[16] John Bond, "Parallel-Processing Concepts Finally Come Together in Real-Time", Computer Design, June 1, 1987, pp.51-70.

[17] Laxmi N. Bhuyan and Dharma P. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Network", IEEE Transactions on Computers, Vol. c-33, No. 4, April 1984, pp. 323-333.

[18] Charles L. Seitz, "The Cosmic Cube", Communications of the ACM, Vol. 28, No. 1, January 1985, pp. 22-33.

APPENDIX A
SSME MATHEMATICAL MODEL

SYMBOLS USED IN THE EQUATIONS

*	multiply
/	divide
fabs	absolute value
sqrt	square root
pow	power
||	logical OR
dt	integral
x	temporary variable
xx	temporary variable

```
 1. OFP1 =  B11*DWFD2/SF1;
 2. PFD1 =  PFS+B12*SF1*SF1*TPfp1(OFP1);
 3. TFP1 =  B13*SF1*SF1*TTfp1(OFP1);
 4. SF1x  =  B14*(TFT1-TFP1);
 5. SF1xx =  SF1x;
 6. if (SF1>ePF1||(TFT1>TBF1 && SF1<ePF1)) dt_SF1=SF1xx; else SF1=0;
 7. PFT1I = P5-B62*DWFT1*fabs(DWFT1)/p5;
 8. CPA = TCp(T5);
 9. CP = CPA+B78+(B84*PFT1I+B88*PFT1I*PFT1I)/(max(B126,T5)-B127);
10. DTHFT1 = sqrt(CP*T5*(1-pow(PFI/P5,B117*(B144-1)/B144)));
11. U_CFT1 = B15*SF1/DTHFT1;
12. TFT1 = B16*DWFT1*DTHFT1*TTft1(U_CFT1);
13. DWFT1 = sqrt(B17*p5*(P5-PFI));
14. OFP2 = B18*DWFD2/SF2;
15. PFD2 = PFD1+B19*SF2*SF2*TPfp2(OFP2);
16. TFP2 = B20*SF2*SF2*TTfp2(OFP2);
17. DWFD2 = (DWFN+DWMC+DWFNBP+B119*PMFVD)/B110;
18. A_AMFV = Tmfv(XMFV);
19. SF2x = B23*(TFT2-TFP2);
20. SF2xx = SF2x;
21. if(SF2>ePF2 || (TFT2>TBF2 && SF2<ePF2)) dt_SF2=SF2xx; else SF2=0;
22. FFP = DWFPOI/(DWFPOI+DWFPF);
23. PRFT2 = pow(PFI/PFP,B8);
24. dFT2 = Td(FFP);
25. DWFT2 = B24*PFP*TPR(sqrt(PFI/PFP),1.0)*
          (B125+B124*SF2+B123*SF2*SF2)/sqrt(TFP);
26. TF2 = TT(FFP);
27. CPF2 = TCpt(FFP);
28. DTHFT2 = sqrt(CPF2*TF2*(1-pow(PRFT2,(dFT2-1)/dFT2)));
29. U_CFT2 = B25*SF2/DTHFT2;
30. TFT2 = B26*DWFT2*DTHFT2*TTft2(U_CFT2);
```

```
31. TFT2DA = TFP-B9*(TFT2*SF2)/(9340.0*CPF2*DWFT2);
32. TFT2Dx = (TFT2DA-TFT2D)/tFT2DA;
33. dt_TFT2D = TFT2Dx;
34. DWCLIx = B1*(PFD2-PCL-B2*fabs(DWCLI)*DWCLI);
35. dt_DWCLI = DWCLIx;
36. DWCLOx = B3*(PCL-PFI-B4*fabs(DWCLO)*DWCLO);
37. dt_DWCLO = DWCLOx;
38. PCLx = B5*(DWCLI-DWCLO);
39. dt_PCL = PCLx;
40. OOP1 = B27*(DWMOV+DWOP3)/SO1;
41. POD1 = POS+B28*SO1*SO1*TPop1(OOP1);
42. DWOSx =  B31*(POT-POS-B32*DWOS);
43. dt_DWOS = DWOSx;
44. POSx = B33*(DWOS-DWMOV-DWOP3);
45. dt_POS = POSx;
46. TOP1 = B34*SO1*SO1*TTop1(OOP1);
47. SO1x = B35*(TOT1-TOP1);
48. SO1xx = SO1x;
49. if (SO1>ePO1 || (TOT1>TBO1 && SO1<ePO1)) dt_SO1=SO1xx; else SO1=0;
50. OOT1 = B38*SO1/DWOT1;
51. TOT1 = B36*DWOT1*DWOT1*TTot1(OOT1);
52. ROT1 = TRot1(OOT1);
53. DWOT1x = B106*(POD2-POD1-(B37+ROT1)*fabs(DWOT1)*DWOT1);
54. dt_DWOT1 = DWOT1x;
55. OOP2 = B39*(DWMOV+DWOT1+DWOP3)/SO2;
56. POD2 = POD1+B40*SO2*SO2*TPop2(OOP2);
57. TOP2 = B41*SO2*SO2*TTop2(OOP2);
58. POTPR = POD2 - 1000.0;
59. SO2x = B42*(TOT2-TOP2-TOP3);
60. SO2xx = SO2x;
61. if (SO2>ePO2 || (TOT2>TBO2 && SO2<ePO2)) dt_SO2=SO2xx; else SO2=0;
62. OOP3 = B43*DWOP3/SO2;
63. POD3 = POD2 + B45*SO2*SO2*TPop3(OOP3);
64. TOP3 = B44*SO2*SO2*TTop3(OOP3);
65. DWOP3x = B46*(POD3-PPOS-B104*DWOP3*DWOP3);
66. dt_DWOP3 = DWOP3x;
67. PROT2 = pow(PFI/POP,B55);
68. FOP = DWOPOI/(DWOPOI+DWOPF);
69. dOT2 = Td(FOP);
70. DWOT2 = B48*POP*TPR(sqrt(PFI/POP),1.0)*
            (B122+B121*SO2+B120*SO2*SO2)/sqrt(TOP);
71. TO2 = TT(FOP);
72. CPO2 = TCpt(FOP);
73. DTHOT2 = sqrt(CPO2*TO2*(1-pow(PROT2,(dOT2-1)/dOT2)));
74. U_COT2 = B47*SO2/DTHOT2;
75. TOT2 = B29*DWOT2*DTHOT2*TTot2(U_COT2);
76. TOT2DA = TOP-B10*(TOT2*SO2)/(9340.0*CPO2*DWOT2);
77. TOT2Dx = (TOT2DA-TOT2D)/tOT2DA;
78. dt_TOT2D = TOT2Dx;
```

```
 79. DWOP2C = sqrt(B115*(POD3-POD1));
 80. PPOSx = B49*(DWOP3-DWFPO-DWOPO-DWOP2C);
 81. dt_PPOS = PPOSx;
 82. DWFPF = B50*(P9-PFP-B51*DWFPF*DWFPF/p9)*STEPTIME+DWFPF-2*DWFPF*CORR;
 83. A_AFPV = Tfpv(XFPV);
 84. DWFPOx = B52*(PPOS-PFP-B53*DWFPO*DWFPO/
              (A_AFPV*A_AFPV)-B54*DWFPOI*DWFPOI);
 85. dt_DWFPO = DWFPOx;
 86. WFPOIx = DWFPO-DWFPOI;
 87. dt_WFPOI = WFPOIx;
 88. EFPO = TEfpo(WFPOI);
 89. DWFPOI = EFPO*DWFPO;
 90. PFPx = B56*(DWFPF+DWFPOI-B111*DWFT2);
 91. PFP = PFPx;
 92. TFP = TF2+B112*T9;
 93. DWOPF = B57*(P9-POP-B58*DWOPF*DWOPF/p9)*STEPTIME+DWOPF-2*DWOPF*CORR;
 94. A_AOPV = Topv(XOPV);
 95. DWOPOx = B59*(PPOS-POP-B61*DWOPO*DWOPO/
              (A_AOPV*A_AOPV)-B60*DWOPOI*DWOPOI);
 96. dt_DWOPO = DWOPOx;
 97. WOPOIx = DWOPO-DWOPOI;
 98. dt_WOPOI = WOPOIx;
 99. EOPO = TEopo(WOPOI);
100. DWOPOI = EOPO*DWOPO;
101. POPx = B63*(DWOPF+DWOPOI-B107*DWOT2);
102. dt_POP = POPx;
103. TOP = TO2+B113*T9;
104. PFIx = B64*(DWFT1+DWOT2+DWFT2-B108*DWFI);
105. dt_PFI = PFIx;
106. DWFIA = B65*PFI*TPR(PC/PFI,0.0)/sqrt(TFI);
107. DWFIx = (DWFIA-DWFI)/tDWFI;
108. dt_DWFI = DWFIx;
109. TFI = B66*TFP+B67*TOP+B68*T5;
110. WOIx = DWMOV-DWOI;
111. dt_WOI = WOIx;
112. A_AMOV = Tmov(XMOV);
113. EOI = TEoi(WOI);
114. DWMOVx = B69*(POD2-PC-B70*DWMOV*DWMOV/(A_AMOV*A_AMOV)-B71*DWOI*DWOI);
115. dt_DWMOV = DWMOVx;
116. DWOI = EOI*DWMOV;
117. PCIES = B114*PC;
118. PCx  = B72*(B109*DWFI+DWOI-DWCN);
119. dt_PC = PCx;
120. FTC = (DWOI+DWOP3)/(DWOI+DWFI);
121. DWCN = B73*PC/C;
122. C = B128+B129*FTC+B130*FTC*FTC;
123. MR = (DWMOV+DWOP3)/DWFD2;
124. TC = TTc(FTC);
125. SU5x = B77/p5*(DQW15+DQW25+B7*DWMC-H5*DW5);
```

```
126. dt_SU5 = SU5x;
127. P5 = TPH2(SU5,p5);
128. H5 = SU5 + P5/(9336.0*p5);
129. DW5 = DWFT1;
130. DWMCx = B75*(PMFVD-P5-B76*fabs(DWMC)*DWMC/p5);
131. dt_DWMC = DWMCx;
132. p5x = B77*(DWMC-DW5);
133. dt_p5 = p5x;
134. T5 = TTH2(SU5,p5);
135. DQW15 = B79*(B131+B132*T5)*(TW15-T5)*pow(fabs(DWMC),0.8);
136. DQW25 = B80*(B131+B132*T5)*(TW25-T5)*pow(fabs(DWMC),0.8);
137. DQTC5 = B81*(TC-TW15)*pow(fabs(DWCN),0.8);
138. TW15x = B82*(DQTC5-DQW15);
139. dt_TW15 = TW15x;
140. TW25x = B82*(-DQW25);
141. dt_TW25 = TW25x;
142. SU4x = B87/p4*(DQW14+DQW24+B6*DWFN-H4*DW4);
143. dt_SU4 = SU4x;
144. DWFNx = B85*(PMFVD-P4-B86*fabs(DWFN)*DWFN);
145. dt_DWFN = DWFNx;
146. P4 = TPH2(SU4,p4);
147. H4 = SU4+P4/(p4*9336.0);
148. p4x = B87*(DWFN-DW4);
149. dt_p4 = p4x;
150. T4 = TH2(SU4,p4);
151. DQW14 = B89*(B131+B132*T4)*(TW14-T4)*pow(fabs(DWFN),0.8);
152. DQW24 = B90*(B131+B132*T4)*(TW24-T4)*pow(fabs(DWFN),0.8);
153. DQTC4 = B91*(TC-TW14)*pow(fabs(DWCN),0.8);
154. TW14x = B92*(DQTC4-DQW14);
155. dt_TW14 = TW14x;
156. TW24x = B93*(-DQW24);
157. dt_TW24 = TW24x;
158. DW4x = B94*(P4-P9-B95*fabs(DW4)*DW4/p4);
159. dt_DW4 = DW4x;
160. P9x = B96*(DW4+DWFNBP-DWOPF-DWFPF);
161. dt_P9 = P9x;
162. PMFVD = PFD2-B22*fabs(DWFD2)*DWFD2/(A_AMFV*A_AMFV);
163. A_ACCV = Tccv(XCCV);
164. T9 = B100*P9/p9;
165. p9 = (DWFNBP+DW4)/(DW4/p4+B105*DWFNBP);
166. DWFNBPx = B98*(PMFVD-P9-B99*fabs(DWFNBP)*DWFNBP/(A_ACCV*A_ACCV));
167. dt_DWFNBP = DWFNBPx;
168. CORR = (p9old-p9)/p9old;
169. QFFM = (1.0-B133*(TFS-B134))*(B135+B136*DWFD2+B137*SF1);
170. TTFP1 = (TFS-B134)+(B145+B146*SF1+B147*SF1*SF1);
```